Programación Python para Machine Learning

Lección 8: Redes Neuronales Artificiales.

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Lección 8: Redes Neuronales Artificiales.

1. Introducción

El cerebro humano lo conforman un inmenso número de neuronas conectadas unas con otras en forma de red. Las neuronas son las encargadas de procesar colectivamente la información sensorial.

En su forma más básica, una neurona consta de dendritas, que reciben la información en forma de impulso, núcleo, que procesa la información, y el axón, por el que sale la información procesada y se conecta con las dendritas de otra neurona.

En esta lección se estudiarán las Redes Neuronales Artificiales, un modelo supervisado no lineal de Machine Learning inspirado en capacidades de aprendizaje del cerebro de los seres humanos.

Las Redes Neuronales se estructuran mediante una compleja arquitectura que se asemeja a la que forman las neuronas en el cerebro humano, dotando al sistema de la capacidad de identificar con un grado alto de precisión patrones dentro de una determinada fuente información disponible.

De este modo, se abordará conceptos como qué son las redes neuronales, sus principios teóricos, y cómo implementarlas en Python.

Objetivos

- Conocer los principios en los que se basan las Redes Neuronales Artificiales.
- Entender los elementos clave que conforman una Red Neuronal.
- Implementar un modelo de Red Neuronal en Python para resolver problemas de clasificación y regresión.
- Identificar los aspectos a tener en cuenta para mejorar el rendimiento de una Red Neuronal.

2. Principios teóricos de las Redes Neuronales

Las Redes Neuronales Artificiales son modelos de Machine Learning que realizan operaciones matemáticas siguiendo una estructura de red. La estructura más común de una red neuronal está formada por capas, cada una de las cuales tiene un número de neuronas o nodos. Cada nodo se denomina perceptrón, por lo que a las redes neuronales también se les denomina Perceptrón Multicapa. Cada neurona realiza una operación matemática simple utilizando la información procedente de las neuronas de la capa anterior, emitiendo el resultado a la siguiente capa.

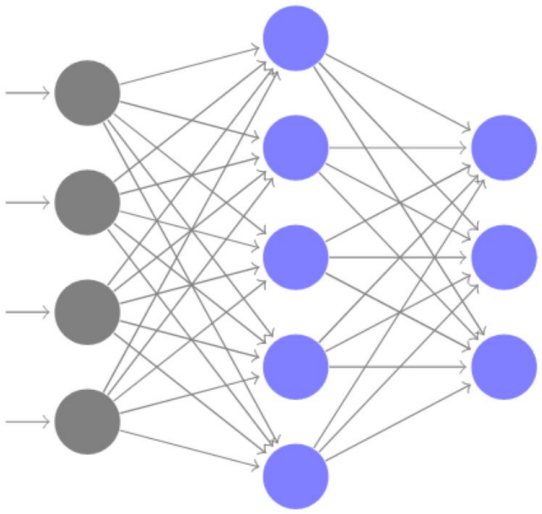

Ilustración 1: Estructura de una Red Neuronal Artificial.

Según el ejemplo que se puede ver en la Ilustración 1, la primera capa de la red neuronal se denomina capa de entrada. Es la encargada de recibir el valor de las variables independientes. La capa intermedia es la denominada capa oculta. La última capa, la capa de salida, combina los valores que salen de la capa intermedia para generar la predicción. Las flechas representan las conexiones entre neuronas, los conocidos como pesos.

Los nodos o perceptrones (véase Ilustración 2) son considerados las unidades funcionales de una red neuronal.

Cada nodo computa la suma ponderada de sus entradas según los pesos de las conexiones y le añade un término bías, para posteriormente aplicar a toda una función de activación.

La introducción de esta función es la que le da al modelo las posibilidades no lineales. La capa de entrada no tiene función de activación.

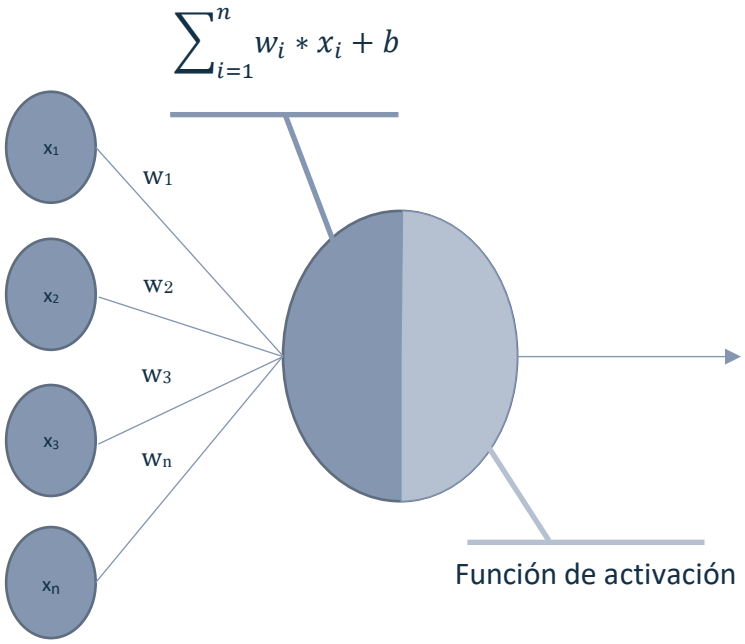

Ilustración 2: Perceptrón

La **función de activación** representa un elemento de control de la información trasciende de una capa a la siguiente. Estas funciones convierten el valor resultante de la combinación de las entradas, pesos y bías, en un nuevo valor.

Las funciones de activación más populares, aunque existen muchas más, son:

- La **función ReLU,** que devuelve la entrada solo si está por encima de cero. Si el valor de entrada está por debajo, el valor de salida es cero.

$$ReLU(x) = \max(0, x)$$

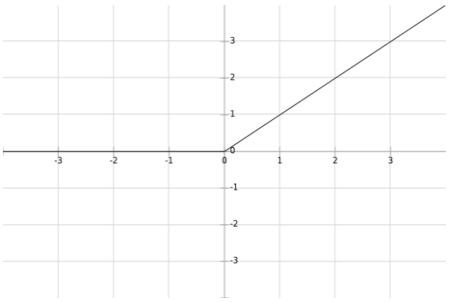

Ilustración 3: Función de activación ReLU.

- La **función sigmoide,** que transforma el valor de entrada al rango (0, 1).

$$Sigmoide(x) = \frac{1}{1 + e^{-x}}$$

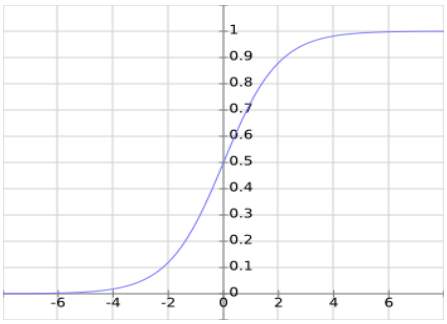

Ilustración 4: Función de activación Sigmoide.

- La **función tangente hiperbólica**, con salida acotada en el rango (-1, 1).

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

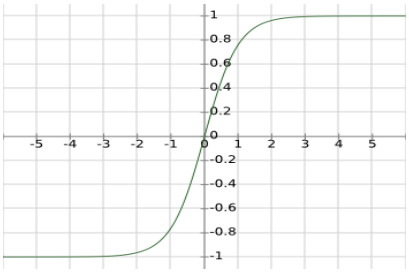

Ilustración 5: Función de activación Tangente hiperbólica.

El entrenamiento de una red neuronal es el proceso consistente en encontrar el valor de los pesos y bías de forma que, las predicciones que se generen a partir de las variables de entrada, sean lo más cercanas posibles al valor esperado en la variable de salida.

El proceso de entrenamiento de una red neuronal es el siguiente:

1. Inicializar una red con valores para pesos y bías de modo aleatorio.
2. Cada patrón de entrenamiento es utilizado para calcular el error que comete la red en la salida.
3. Determinar el grado con el que cada peso y bías a influenciado en el error de la predicción.
4. Alterar los pesos de la red de modo proporcional a su responsabilidad en el error.

Se trata un proceso iterativo en que se repiten los pasos 2, 3, 4 hasta alcanzar un criterio de parada. El algoritmo de **retropropagación** junto con la **optimización por descenso de gradiente** son el método más utilizado para llevar a cabo el proceso.

La **función de pérdida** o función de coste es la métrica que cuantifica la distancia entre el valor real y el valor predicho por la red, es decir, mide el error al realizar predicciones.

Cuanto más próximo a cero, mejor es la capacidad predictiva de la red. La función de pérdida puede calcularse para una única instancia o para un conjunto de datos.

Atendiendo al tipo de problema, se hace necesario utilizar una función u otra de pérdida. Para problemas de regresión, la que tiene más sentido utilizar es el error cuadrático medio. Para problemas de clasificación utilizarse la función log loss.

3. Implementación en Python de una Red Neuronal

Las Redes Neuronales presentan una versatilidad dentro del Machine Learning que las hacen ser un modelo apropiado para resolver tanto problemas de clasificación como problemas de regresión.

Se puede llevar a cabo la construcción de un modelo de redes neuronales en Python utilizando el módulo `neural_network` de scikit-learn.

En todo caso, hay que recordar que cuando se trata con un modelo de Redes Neuronales, todas las variables deben ser de numéricas.

Por tanto, toda variable categórica en el conjunto de datos debe ser convertidas en numéricas del modo más adecuado y representativo posible.

3.1. Redes Neuronales para clasificación

Scikit-Learn cuenta con la clase `MLPClassifier` dentro de su módulo `neural_network`, que permite crear una red neuronal para proceder con problemas de clasificación. Se hará uso del conjunto de datos *magic* del repositorio *UCI Machine Learning.*

```
from pandas import read_csv
from sklearn.neural_network import MLPClassifier
import time, random
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score,
balanced_accuracy_score

seed=random.seed(time.time())
filename = '../datasets/magic04.data'

col_names=['fLength','fWidth','fSize','fConc','fConc1','fAsym','fM3Lon
g','fM3Trans','fAlpha','fDist','class']
data = read_csv(filename, names=col_names)

X = data[data.columns[:-1]]
Y = data['class']

X_train, X_test, y_train, y_test = train_test_split(X_train, y_train,
test_size = 0.3, random_state=seed)
```

```
scaler =StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test= scaler.transform(X_test)

model = MLPClassifier(hidden_layer_sizes=(100,),
max_iter=1000).fit(X_train, y_train)
y_pred = model.predict(X_test)

bacc =  balanced_accuracy_score(y_pred, y_test)
acc =  accuracy_score(y_pred, y_test)
cm =  confusion_matrix(y_pred, y_test)

print(bacc, acc)
print(cm)
```

Teniendo en cuenta sus principios teóricos, para configurar una Red Neuronal de modo adecuado para un problema de clasificación, se deben tener en cuenta una serie de aspectos clave, que se pueden configurar a través de los parámetros de la clase `MLPClassifier`:

- **Número y tamaño de las capas ocultas**: Se realiza a través del parámetro `hidden_layer_sizes` que utilizando una tupla de enteros determina el número de capas ocultas (tamaño de la tupla) y el número de nodos por capa de la red. La capa de entrada siempre tendrá el número de dimensiones del problema y la de salida el número de clases del problema.
- La **función de activación**: Por defecto, la clase `MLPClassifier` tiene configurada la función de activación ReLU. El parámetro `activation` permite cambiar esta opción.
- El **algoritmo de entrenamiento**: Por defecto, la clase `MLPClassifier` tiene configurada el algoritmo Adam, un método de descenso de gradiente estocástico. Este tiene buen rendimiento ante conjuntos de datos con muchas instancias. Para conjuntos más pequeños, scikit-learn trae implementado L-BFGS. El parámetro `solver` permite cambiar esta opción.
- La **tasa de entrenamiento**: Controla el tamaño del paso en la actualización de los pesos. El parámetro que lo establece es `learning_rate_init` de tipo doble y por defecto con un valor de 0.001. Puede ser constante o dinámica.

- Número máximo de **épocas de entrenamiento**: Número de ciclos máximo del algoritmo de entrenamiento. El parámetro `max_iter` por defecto está inicializado a 200, aunque puede ser modificado.

3.2. Redes Neuronales para regresión

Scikit-Learn cuenta con la clase `MLPRegressor` dentro de su módulo `neural_network`, que permite crear una red neuronal para proceder con problemas de clasificación. Se hará uso del conjunto de datos *housing* del repositorio *UCI Machine Learning.*

```
from pandas import read_csv
from sklearn.neural_network import MLPRegressor
import time, random, pandas, numpy
from sklearn.preprocessing import RobustScaler,MinMaxScaler,
StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

seed=random.seed(time.time())
filename = '../datasets/housing.csv'

col_names=['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
data = read_csv(filename, names=col_names,sep=" ")

X = data[data.columns[:-1]]
Y = data['MEDV']

#X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size =
0.5, random_state=seed, stratify=Y)
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size =
0.3, random_state=seed)

scaler =StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test= scaler.transform(X_test)

model = MLPRegressor(hidden_layer_sizes=(100,), max_iter=3000,
activation='relu').fit(X_train, y_train)
y_pred = model.predict(X_test)


rmse= numpy.sqrt(mean_squared_error(y_test, y_pred))

print(rmse)

plt.plot(model.loss_curve_)
```

Aunque los principios teóricos que determinan la configuración de una Red Neuronal de modo adecuado para un problema de regresión son los mismos que para los problemas de clasificación, se deben tener en cuenta una serie de aspectos clave que pueden cambiar y que se pueden configurar a través de los parámetros de la clase `MLPRegressor`:

- **Número y tamaño de las capas ocultas**: Idéntico que en el modelo de clasificación.
- La **función de activación**: Por defecto, la clase `MLPRegressor` tiene configurada la función de activación ReLU. El parámetro `activation` permite cambiar esta opción, aunque ReLU suele ser la que mejor rendimiento tiene en este tipo de problemas.
- El **algoritmo de entrenamiento**: Idéntico que en el modelo de clasificación.
- La **tasa de entrenamiento**: Idéntico que en el modelo de clasificación.
- Número máximo de **épocas de entrenamiento**: Idéntico que en el modelo de clasificación.

4. Consejos prácticos sobre las Redes Neuronales

4.1. Ajuste de parámetros

El rendimiento de un modelo predictivo implementado mediante Redes Neuronales está tremendamente condicionado al ajuste de sus parámetros. En particular, hay una serie de aspectos que deben ser tenidos muy en cuenta debido a la sensibilidad que tiene el algoritmo ante ellos:

1. La topología de la red, es decir, el número de capas ocultas y el número de nodos en cada capa.
2. La tasa de entrenamiento.
3. El número máximo de épocas.
4. El momento, solo en caso del solver 'sgd'.

No cabe más posibilidad que completar este proceso mediante la validación mediante una búsqueda en rejilla o *grid*.

4.2. Clasificación multiclase

Por su naturaleza, las Redes Neuronales están diseñadas para poder resolver problemas de clasificación binarios, problemas de regresión y problemas de clasificación multiclase. Por defecto, la clase `MLPClassifier` de scikit-learn trae configurada la posibilidad de acometer la clasificación multiclase de manera que queda totalmente transparente al programador esta característica.

4.3. Consideraciones finales a tener en cuenta

Una de las propiedades más importantes de las Redes Neuronales es que tienen un tremendo potencial a la hora de aprender modelos no lineales. Sin embargo, la función de pérdida que implementan es no convexa, por lo que existe más de un mínimo local en los que puede caer su optimización. Esto conlleva que no siempre haya una estabilidad en su rendimiento.

A su vez, se requiere el ajuste minucioso para cada conjunto de datos de hiperparámetros como la topología de la red, la tasa de entrenamiento o las épocas del entrenamiento, debido a la sensibilidad que presenta ante ellos. Igualmente, estos modelos son sensibles al escalado de características.

Por último, se tiene la posibilidad de aprender modelos en tiempo real (aprendizaje *on-line*), en el que, una vez entrenado el modelo, este puede ser actualizado con nuevos patrones de datos que pudieran estar disponibles.

5. PUNTOS CLAVE

En esta lección se ha aprendido:

- Conocer los conceptos generales de las Redes Neuronales Artificiales.
- Detallar los principios y los elementos fundamentales que conforman una Red Neuronal
- Implementar en scikit-learn una Red Neuronal tanto para problemas de clasificación como para regresión.
- Profundizar en aquellos aspectos de una Red Neuronal que pueden ser determinantes a la hora de influir en su rendimiento.